53341O

AN EXPERIMENT IN SCIENTIFIC PROGRAM UNDERSTANDING

Mark E. M. Stewart
Dynacs Engineering, Inc.
2001 Aerospace Parkway
Brook Park, OH 44142

Abstract

This paper concerns a procedure that analyzes aspects of the meaning or semantics of scientific and engineering code. This procedure involves taking a user's existing code, adding semantic declarations for some primitive variables, and parsing this annotated code using multiple, independent expert parsers. These semantic parsers encode domain knowledge and recognize formulae in different disciplines including physics, numerical methods, mathematics, and geometry. The parsers will automatically recognize and document some static, semantic concepts and help locate some program semantic errors. Results are shown for three intensively studied codes and seven blind test cases; all test cases are state of the art scientific codes. These techniques may apply to a wider range of scientific codes. If so, the techniques could reduce the time, risk, and effort required to develop and modify scientific codes.

Introduction

Scientific software development consumes scarce resources and limits the pace of science and engineering. Code development is slow because, in part, it has not been possible to automate the comprehension of scientific and engineering software, and comprehension is a prerequisite for performing several time-consuming, error-prone software development tasks.

However, the classical notation and methods of mathematics and physics are the knowledge representation for comprehending scientific code. Scientific programs involve an organization of these equations and concepts. The programs from a wide range of scientific and engineering fields use and reuse these fundamental concepts in different combinations. This paper explains an experiment in representing, recognizing, and checking these fundamental scientific semantics. Two practical scientific software problems motivate this experiment: locating semantic errors in code, and scientific code documentation.
Reducing the errors in a scientific or engineering program until its results are trusted involves ensuring the program's semantics are correct. The existing software development tools (lint, ftnchek, make, dbx, SCCS, call tree graphs, memory leak testing) do not fully alleviate this problem and deal only superficially with semantics. Further, scientific code verification techniques (comparison with available analytic and experimental results; verification of theoretical properties--convergence and order of accuracy) can only detect the presence of an error; *finding* this error often leads to a time-consuming manual search. For example, the second-difference code (1) contains a geometrical error in the grid index I, which is exceedingly hard to find manually.

$$FS(I,J) = DW(I+2,J) - 2.*DW(I,J) + DW(I-1,J) \quad (1)$$

However, it can be found automatically with this semantic analysis procedure.

The second problem motivating this work is that understanding another programmer's code is usually frustrating and time consuming, and to understand code well enough to modify it confidently requires a large time investment. Suggestive variable names, program comments, program manuals, and communications with the developer are means to convey an understanding of a code, but these methods are often neither adequate nor efficient. This semantic analysis procedure can represent and recognize important code details.

Modern programming practice attempts to reduce the number of code development errors and to ease code modification. Software reuse (through subroutine libraries) and object-oriented programming also target these problems. Further, modern programming tools (lint, ftnchek, memory-leak testing) use the programming language's semantics to identify some errors. Recently there has been work in high-level specification languages[1] where a symbolic manipulation program (Maple, Mathematica) is used to write subroutines or even programs. The field of formal methods[2] attempts to solve these software problems by using logic, set theory, functions, and algebra to develop mathematical models for systems and to rigorously prove code properties.

This is a preprint or reprint of a paper intended for presentation at a conference. Because changes may be made before formal publication, this is made available with the understanding that it will not be cited or reproduced without the permission of the author.

Previous efforts in program comprehension exist. Martino[3] considers automated program parallelization. Wills[4] uses a representation and parsing to recognize algorithmic elements and data structures. (The current work emphasizes scientific domain knowledge, declaration of input properties, and procedure evaluation with real world test cases.) In natural language understanding, parsing has been combined with an ontology to recognize and represent the semantics of written text[5].

The limitations of these existing tools, the application domains of these previous approaches, and the cost of manual semantic analysis are the motivations for the current experiment. As a complementary tool, automated semantic analysis could reduce the time, risk, and effort during original code development, subsequent maintenance, second party modification, and reverse engineering of undocumented code.

This paper follows experimental report form with a thesis, procedure, results, discussion, and conclusion.

Thesis

The thesis of this semantic analysis experiment is that fundamental physical and mathematical formulae and concepts are reused and reorganized in scientific and engineering codes. This domain knowledge can be represented in parsers[6,7,8], and when combined with other methods, these parsers can recognize scientific code semantics.

If this experiment in automated analysis succeeds, the resulting tool would help locate errors during code development and document code for modification.

Procedure

In outline, the current procedure for testing this thesis consists of four key stages. First, the user adds **semantic declarations** to his/her existing FORTRAN program (2).

```
C?  MA == mass
C?  ACC == acceleration
    FF = MA*ACC                    (2)
```

Distinguished by "C?" these declarations provide the mathematical or physical identity of primitive variables in the user's program. Second, the procedure syntactically parses the user's program into a data structure representation.

Third, a translation scheme converts statements in the user's FORTRAN program into statements in different context languages. For example, the FORTRAN expression in (2) is converted to the physical dimensions expression (3) and the physical quantity expression (4).

```
(M) * ( L*T**-2 )                  (3)
mass * acceleration                (4)
```

These context languages reflect the different **aspects** of program statements that scientists and engineers analyze. Aspects include mathematical or physical quantity, geometrical (grid) location, geometrical entity, vector entity, dimensions, units, array references, and array assignments.

Fourth, independent expert parsers examine the translated phrases and attempt to recognize formulae from their area of expertise. For example, a dynamics expert parser would include the rule (5), be able to recognize the phrase (4) as Newton's law, infer (4) is "force", and assign this result to FF in (2).

```
force : mass * acceleration        (5)
```

Further, the units expert parser can reduce (3) and verify units. The other expert parsers act similarly (see Table 1).

[illegible]	[illegible]	[illegible]	[illegible]
Quantity-Math	5	772	72
Quantity-Physical	3	766	114
Value / Interval	2	223	27
Grid Location	4	1801	235
Geometrical Entity	1	447	20
Vector Entity	1	300	15
Non-Dimensional	1	72	5
Dimensions	1	59	10
Units	1	71	14
Object Analysis	1	128	10
Array Analysis	2	121	3

Table 1: Aspect analyses performed by the semantic analysis procedure including number of parsers for each aspect, number of Yacc[7] parser rules, and fundamental equations. Rule (5) is a fundamental equation; some equations require several parser rules.

When an expert parser recognizes an expression, it annotates the parse tree. Further, a graphical user interface (GUI) displays the annotated parse tree as shown in Figure 1. The user may point to variables and expressions in his/her code, and the GUI displays any semantic interpretation and its derivation. In Figure 1 the highlighted expression is recognized as density. The GUI highlights recognized errors, undefined quantities, and unrecognizable expressions. Further, the GUI provides detailed scientific and technical definitions and explanations.

```
File  Dictionary  Metrics  Highlight  Language  About

c
c........determine inlet static temperature from isentropic relations
    tsrat - gasisn(emach1, 2, gam)
    tsin - tsrat*t0in
C?  TSIN -- TEMPERATURE_ABSOLUTE
    atsin(i) - tsin - dftodr
c
c........determine inlet density, velocity, viscosity, Reynolds number
    rhoin - psin*144/(rgas*tsin)
    uin - emach1*sqrt(gam*rgas*tsin)
    arhoui(i) - rhoin*uin
    auin(i) - uin
    visin - visref*(tsin/tvisrf)**vispwr
    recx1 - rhoin*uin*chordx/visin
    arecx1(i) - recx1
c
c........determine inlet thermal conductivity and Prandtl number
    conin - conref*(tsin/tconrf)**conpwr
    prndl1 - visin*cepe/(conin*777.649)
c

Quantity: DENSITY                          Metascope    Undefined
Location: UNKNOWN                          Microscope   Error
Dimensions: length**-3 mass**1             Back         Not Understood
Units: slugs/ft3                           Fwd          Performance
Accuracy:

Deduced from equation:
DENSITY - PRESSURE / WORK_PUM

Expertise: GASDYNAMICS
File: flow_inlet.f     Undefined: 35     Errors: 0     Not Understood: 7

'DENSITY'
 The mass of a region of space divided by its volume.

'DERIV'
 The discrete derivative of one variable with respect to another (ratio of two DELTAs).
 This symbol takes two adjectives: the function (numerator) and the variable (denominator).

'DERIV2'
 The discrete second derivative of one variable with respect to two others.
 This symbol takes three adjectives: the function (numerator) and the first and second
```

Figure 1: GUI display for the semantic analysis program. The top window displays a user's code; variables and expressions may be selected for explanation. The middle region explains this selected text. In this case, the physical quantity is density, it does not have a grid location, and it has the displayed dimensions, units, and derivation. The bottom region displays the semantic dictionary/lexicon.

Extended Example of Parsing for Code Recognition

The most important step in the above procedure is the expert parsers' analysis. An example of how parser rules operate to recognize an expression is instructive. To infer the meaning of the variable VAR in (6), the parser sequentially examines the right-hand side of (6). When the parser reads *energy<internal>* (EI), it anticipates rule (7d) and the tokens '+' and *work*.

```
C? P == pressure<static>, RHO == density<static>
C? V == speed, EI == energy<internal>
C
      VAR = EI + P / RHO + 0.5 * V * V                (6)
```

When the parser reads *pressure<static>* (P), it expects rule (7c) to produce *work* and the tokens '/' and *density<static>* to appear. When the parser sees all the tokens of rule (7c), it reduces them to the token *work*, and when all the tokens of rule (7d) are present, they are reduced to the token *enthalpy*. The parser anticipates rule (7f) next. Similarly, the parser recognizes rules (7b), (7a), and (7f) to infer that the expression and VAR represent *enthalpy<total>*.

```
speed_squared        : speed * speed                        (7a)
energy<kinetic>      : half * speed_squared                 (b)
work                 : pressure<static>/density<static>     (c)
enthalpy             : energy<internal> + work              (d)
sound_speed_squared  : gamma * work                         (e)
enthalpy<total>      : enthalpy + energy<kinetic>           (f)
```

(These rules have been simplified to intensive quantities.)

Parser rules (7) are automatically converted to a subroutine by the program Yacc[7]. Table 2 gives a flavor of the expert parser rules.

Properties of the Procedure

Several additional features and properties of this automated semantic analysis procedure deserve mention: semantic declaration terms, mathematical rules, the generality of recognition, the nature of error detection, and the presence of ambiguities.

Terms in Semantic Declarations

The code fragment (6) defines primitive variables in four semantic declarations. The six defining terms *pressure*, *static*, *density*, *speed*, *energy*, and *internal* are from a lexicon of terms (currently 440), and they closely resemble English technical terms. Further, the knowledge representation uses adjective terms, such as *static* in *pressure<static>*, to modify terms and reduce their number. Multiple adjectives are possible. For example, the term *derivative* takes two adjectives, *derivative<pressure time>*, to represent $\partial p/\partial t$.

Mathematical Rules

The physical rules (7) differ from mathematical rules, since mathematical rules apply to any mathematical or physical quantity. For example, to detect a discrete difference, Δq, the pattern is *variable–variable* where *variable* is any quantity. This pattern matches excessively, and when it does match, additional code compares the aspects of the two variables. If the variables are identical, then the expression is zero; if the variables differ in location only, then the expression is a discrete difference, Δq. The generality and multiple aspect nature of mathematical rules make them relatively hard to develop, but exceedingly useful.

Generality of Recognition

In what forms can a formula appear? The general derivation of equations is dependent upon the fundamental physical equations, the algebraic properties of mathematical expressions (Commutative, Distributive and Associative laws), and the transformation laws of equations (including equation substitution, simplification, and solve). This general derivation of equations implies a non-trivial and potentially expensive search to recognize all possible forms of a formula.

Further, the rules within a LALR(1)[6,7] grammar *simply* reduce one or more input tokens to a single token. Consequently, most transformations, including substitution and commutation, cannot be implemented in a parser.

Consequently, these expert parsers are supplemented with five methods. The first three of these methods give a moderate algebraic search. This strategy tries to avoid expensive general searches each time the formula is encountered. The existing evidence indicates this is a reasonable choice.

In the first method, the experts are applied incrementally to expression components. In (8) the sub-expressions p/ρ, u^2, and $\frac{1}{2}u^2$ are separately referred to each of the expert parsers for analysis.

$$p/\rho + \tfrac{1}{2}u^2 \qquad (8)$$

Second, between these calls to the expert parsers, methods are applied that commute, associate, and distribute (and inverse distribute) the expression.

Mathematical, Numerical Quantity	**Physical Quantity**	**Physical Quantity**
$q \Leftarrow q + 0$	$p \Leftarrow F / \text{area}$	$^{o}C \Leftarrow {}^{o}K - 273.15$
$q \Leftarrow q * 1$	$F \Leftarrow m * A$	$^{o}F \Leftarrow 1.8 * {}^{o}C + 32$
$1 \Leftarrow q / q$	$W \Leftarrow F * \text{length}$	$E \Leftarrow F / q$
$0 \Leftarrow q_1 - q_2$	$E_k \Leftarrow ½ * m * U^2$	$V \Leftarrow I R$
$\Delta q \Leftarrow q_1 - q_2$	$R_u \Leftarrow k * N_A$	$\nu \Leftarrow \mu / \rho$
Polynomials	$R \Leftarrow R_u / \text{Mol. wt.}$	$Pr \Leftarrow C_p \; \mu / k$
$\Sigma q \Leftarrow q + q + ...$	$R \Leftarrow C_p - C_v$	$\text{Reynolds} \Leftarrow \rho * U * \text{length}/\mu$
$q^2 \Leftarrow q * q$	$C_p \Leftarrow \Sigma \text{ (Mass Fract.* } C_p)$	$u*\partial u/\partial x - (1/\rho)*\partial p/\partial x$
$2q \Leftarrow q_1 + q_2$	$\gamma \Leftarrow C_p / C_v$	$U_\theta \Leftarrow r \Omega$
$\Delta^2 q \Leftarrow q - 2q + q$	$w \Leftarrow p / \rho$	$(\partial m/\partial t)_{corr} \Leftarrow \partial m/\partial t \sqrt{\theta} / \delta$
$\partial q/\partial x \Leftarrow \Delta q / \Delta x$	$c^2 \Leftarrow \gamma * p / \rho$	$\text{Circum} \Leftarrow 2 \pi r$
$\partial^2 q/\partial x^2 \Leftarrow \Delta^2 q / \Delta^2 x$	$p / \rho \Leftarrow R * T$	$\text{vol} \Leftarrow \text{length} * \text{area}$
$\partial q/\partial y \Leftarrow \partial q/\partial x * \partial x/\partial y$	$e_i \Leftarrow 1/(\gamma-1) * p / \rho$	$\text{area} \Leftarrow \text{length} * \text{length}$
$\nabla \cdot \mathbf{q} \Leftarrow \text{expression}$	$e_k \Leftarrow ½ * U^2$	**Grid Location, Geometrical Entity**
$\nabla \times \mathbf{q} \Leftarrow \text{expression}$	$h \Leftarrow e_i + w$	
$\nabla^2 q \Leftarrow \text{expression}$	$h_o \Leftarrow h + e_k$	$l \Leftarrow l_1 \pm l_2$
$\mathbf{q_1} \cdot \mathbf{q_2} \Leftarrow \text{expression}$	$M \Leftarrow U / c$	$l \Leftarrow l_1 */ l_2$
$\mathbf{q_1} \times \mathbf{q_2} \Leftarrow \text{expression}$	$\partial m/\partial t \Leftarrow \rho * U * A$	$g \Leftarrow g_1 \pm g_2$
$\text{Jacobian} \Leftarrow \text{expression}$	$P \Leftarrow \text{const} * T^{\gamma/\gamma-1}$	$g \Leftarrow g_1 */ g_2$
Number Value, Number Interval	**Vector Entity**	**Non-Dimensionalization, Dimensions, Units**
$n \Leftarrow n_1 \pm n_2$	$v \Leftarrow v_1 \pm v_2$	$D \Leftarrow D_1 \pm*/ D_2$
$n \Leftarrow n_1 */ n_2$	$v \Leftarrow v_1 */ \text{scalar}$	$D \Leftarrow \text{ftn}(D_1)$
$n \Leftarrow n_1 ** n_2$	$\text{surface} \Leftarrow v_1 * v_2$	$d \Leftarrow d_1 \pm*/ d_2$
$n \Leftarrow \text{ftn}(n_1)$	$\text{scalar} \Leftarrow \text{scalar} \pm \text{scalar}$	$d \Leftarrow \text{ftn}(d_1)$
$r \Leftarrow r_1 \pm r_2$	$\text{scalar} \Leftarrow \text{scalar} */ \text{scalar}$	$u \Leftarrow u_1 \pm*/ u_2$
$r \Leftarrow r_1 */ r_2$	$\text{scalar} \Leftarrow \text{Dot Product}$	$u \Leftarrow \text{ftn}(u_1)$
q = Math/Numerical Quantity; l = Grid Location; g = Geometrical Entity; v = Vector Entity; n = Number Value; r = Number Interval; D = Non-Dimensionalization; d = Dimensions; u = Units		

Table 2: A sampling of expert parser rules used in the semantic analysis method. Many rules are condensed. Due to decomposition a single operation may involve multiple independent aspects (units, grid location and quantity for *x_coordinate – x_coordinate*), and several rules from this table can apply to it.

Third, some limited substitutions can be performed between referrals to the expert parsers. Examples include: $2a_i \Leftarrow a_{i+1} + a_{i-1}$ and the normalizing transformation $u^2 \Leftarrow u*u$.

A fourth method of enhancing recognition is **decomposition** of semantic knowledge. For example, by analyzing a quantity's axis, the vector entity expert parser can recognize a dot product (9) almost independently of the physical quantity, u. (Note: verifying that u_x^2, u_y^2, and u_z^2 are otherwise identical is the further necessary test.)

$$\mathbf{u} \cdot \mathbf{u} \Leftarrow u_x^2 + u_y^2 + u_z^2 \qquad (9)$$

This approximate independence of aspects extends to the analysis of mathematical/physical quantity, number value/interval, grid location, geometrical entity, vector entity, non-dimensionalization, dimensions, units, and object.

In the fifth method, constants are identified from context. For example, in (10) the constant 32 is distinguished from other constants for the parser analyzing temperature equations.

$$^{o}F = 1.8 * {}^{o}C + 32 \qquad (10)$$

The use of 32 in (10) is an example of how this enhanced recognition process is context sensitive[6], since in other contexts this constant can have other meanings. The identification of unit conversion constants is also context sensitive.

Recognition Search Limitations

Despite these enhancements, only a limited recognition search is performed, and practical limitations exist. For example, neither simplifications nor evaluated number expressions can be recognized. Generally these limitations can be avoided by rewriting code using a particular style of structured programming.

For example, the code (11) is not recognizable from stored rules as c ρ_0,

```
C1 = (gamma(J)/rgc)**0.5*p1t0/t1t0**0.5      (11)
```

$$c\,\rho_0 = (\gamma R T_0)^{1/2}\, P_0/(R T_0) = (\gamma/R)^{1/2}\,(P_0/T_0^{1/2}) \quad (12)$$

(gamma, γ are ratio of specific heats; rgc, R are gas constant; p1t0, P_0 are total pressure; t1t0, T_0 are total temperature; c sound speed; ρ_0 total density)

since its derivation (12) involves simplification of the R and T_0 terms. However, it could be recognized if the code were re-written without this cancellation, or if a specific rule were added.

Similarly, the real constants in (13) are evaluations of $\gamma/(\gamma-1)$, $2/\gamma$, and $(\gamma-1)/\gamma$ for a particular value of the ratio of specific heats, γ, namely γ=1.354.

```
numer=3.8249*(r**1.4771)*(1-r**0.26145)*(1-beta**4)
denom= (1-r)*(1-(beta**4)*r**1.4771)                (13)
```

Evaluated number expressions are hard to recognize manually, and to be recognized, this code would have to be rewritten with a semantically declared variable for γ.

Error Detection

This semantic analysis procedure detects errors with direct tests of some code aspects including dimensions, units, and non-dimensionalization. For other code aspects, including mathematical/physical quantity, the semantic analysis procedure attempts to recognize formulae. Unrecognized code may be incorrect or a correct formula beyond the scope of the stored rules. For example, the procedure would declare the aerodynamics equation (14) unrecognized (pressure is incorrectly calculated from density, total energy, velocities and the ratio of specific heats); (14) cannot be declared in error since it may be an unknown rule.

```
P = RHO*(E – ( U*U + V*V ))*(GAM-1)      (14)
```

A novice cannot declare this statement an error either; however, a human expert would combine a fluency in aerodynamics formulae with knowledge of the code's domain to recognize an error.

In cases where multiple conditions must be satisfied before recognition, the failure of one condition is evidence of an error. For example, to recognize a second-difference, formula and geometrical conditions must be satisfied; however, in (1) the geometrical condition is not satisfied and an error is suspected. Although this semantic analysis procedure is not a direct error tester, by reviewing the analysis, users can identify errors relatively easily.

Code Ambiguities

A further theoretical issue is that ambiguities exist in the static analysis of scientific code. The final identity of the variable P is ambiguous in (15).

```
C?  P == pressure<static>
C?  RHO == density
    CC = GAM * P / RHO          (15)
    ...
    P = 1
```

After P is assigned the indistinct value 1, it may still represent pressure, or it may represent something else with the constant value 1. The procedure resolves this ambiguity by assuming no re-use of the variable; the new number value is set, and a warning is generated. The ambiguity can be resolved if the user rewrites the code so that P is not re-used. This ambiguity would not exist if the assignment were P = RHO or P = 3.14 (π) where re-use of the variable P is apparent.

Another ambiguity can exist when deducing an array's layout. In a static analysis, the indices in array assignment can be under-specified and suggest different array layouts. In (16), it is not clear if $N<3$, $N>3$, or $N=3$ since the value of N is not known.

```
C?  N == number<species>
C?  R == radius
    DIMENSION A(10)             (16)
    A(3) = 0.
    A(N) = R
```

This ambiguity is resolved with semantic declarations for array structure or by assuming (and noting) a case.

Rule Ambiguities

When devising semantic rules (Table 2), the ideal is that no more than one rule should apply in any case. Mathematics' and physics' relatively unambiguous character, highly specific semantic terms and patterns, and careful design all help achieve this ideal. However, experience shows a few cases exist where this is not the case. In the current rule design, 12 and 60 in (17) can

```
C? R      == radius<>@ 'inches'
C? RPMC == velocity_angular<angular>@ 'rpm'

   PI = 3.1415                                    (17)
   V = 2.*PI*R/12./60.* RPMC
```

be the unit conversion constants 12 in/ft and 60 sec/min or they can be simplified, 12*60 = 720, and left unexplained. Which of these two rules fire depends on expression order, since unit conversion constants are context sensitive, that is, the units to be converted must be visible elsewhere in the expression.

Equation (18) shows another rule ambiguity. Due to indistinct SI unit conversion factors, the unit conversion constants are ambiguous.

```
C? M == mass@ gram
C? V == velocity @'cm/s'                          (18)
```

$$P = M * V * 10^{-3} * 10^{-2}$$

In (18) the constants are probably 10^{-3} kg/g and 10^{-2} m/cm.

Results

The results take three forms: the recognition of code semantics, the generality of this recognition capability, and measurements of the inference tree.

Recognition of Code Semantics

The results of analyzing three development codes demonstrate feasibility for code semantic recognition. One program, flow_inlet[9], performs data reduction for experimental data analysis (Figure 1); ALLSPD[10] is a three-dimensional chemically reacting fluid flow code; COMDES[11] uses classical aerodynamics for one-dimensional compressor analysis. In these development test cases, the procedure developer devises and tests expert parser rules, and corrects errors. Hence they are not blind test cases.

Highlighted in the GUI of Figure 1 is a recognized expression from the first development code. Other recognized formulae include temperature formulae, viscosity and thermal conductivity calculated from the power law, Reynolds number, and Prandtl number. Most of the not-understood code corresponds to program variables that are defined by function calls and logical expressions. The semantic analysis coding needed for these cases has not yet been developed.

Recognition Metric

To measure this recognition of code semantics, the parse tree is searched for each operation, $a \otimes b$ where $\otimes \in \{ +, -, *, /, ** \}$, intrinsic function reference, ftn(a), and array reference, a(i,j,k). The recognition rate is the fraction of these operations/references where the mathematical/physical quantity is understood. Expression (19) contains 7 operations, 1 intrinsic function reference, and 1 array reference.

```
sqrt(gam(J)*RG*32.174*T0) * (P0/(RG*32.174*T0))
                                                (19)
```

(gam is the ratio of specific heats, RG gas constant, T0 total temperature, and P0 total pressure)

Understanding the first half of the expression as the speed of sound contributes 5/9 to the understanding rate, and recognizing the second half as density contributes 3/9. Since "sound speed * density" is not reducible by the rules, the recognition rate for this expression would be 8/9. If this expression were used subsequently in an expression that is understood, this recognition fraction would be amended to 1.

This recognition metric has limitations. The metric does not compensate for either the restrictiveness of the rules (ie. Δq, Δx must have the same grid location before $\partial q/\partial x \Leftarrow \Delta q / \Delta x$) or attention to comprehension details (ie. boundary conditions recognized in arrays). Further, physical or mathematical quantity is the most valuable information for a user, yet it is the hardest aspect (Table 1) to recognize. Consequently, this is a demanding metric.

In Figure 2, this recognition metric for the ALLSPD[10] development case is plotted against an increasing number of semantic declarations. The curve is offset from the origin since some trivial expressions are recognizable without semantic declarations. Table 3 gives recognition results for the other test cases.

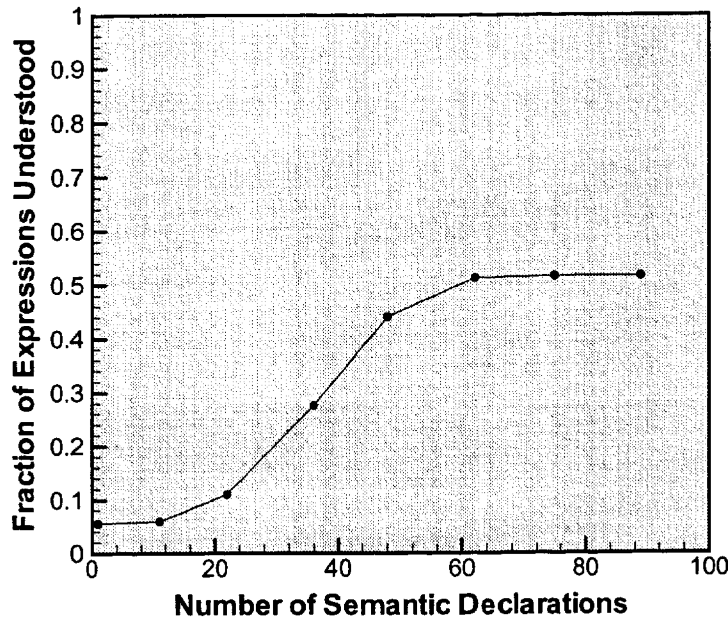


Figure 2: Graph showing the increase in expression understanding as semantic declarations are added to twenty subroutines from the ALLSPD code. The subroutines contain 5278 non-comment FORTRAN statements and 3431 operations to understand. Further work will increase the understanding fraction. The analysis results reflect the analysis code's quality and not the quality or ability of the ALLSPD code.

Generality of Recognition Capability

The generality of this semantic recognition procedure is measured by analyzing seven blind test cases (Table 3). Further, these results demonstrate what a user with a general code might currently expect (after declaring input and primitive variables, but not coding additional mathematical and physical rules). These test cases are one-, two-, and three-dimensional computational fluid dynamics (CFD) codes for turbomachinery problems, and they are typical of CFD codes. The procedure developer examined these test codes to determine semantic declarations for coordinates, solution variables, and other primitive variables. The semantic analysis program was not modified to correct rule omissions.

Recognition results for these blind test cases demonstrate a general semantic recognition capability. Higher recognition rates are expected in the development codes since the procedure developer corrects expert parser errors found during development. Additional work on the procedure will improve these preliminary results. Of course, these results reflect the analysis code's level of development and not the quality of these blind test cases.

Although human scientific software comprehension probably involves a more sophisticated analysis procedure than this, a comparison based on experience is insightful. A code's developer should have a high quantity recognition rate (>80%), while an experienced scientific programmer could recognize 20%-80% of a code depending on their familiarity with the general scientific field, the numerical methods and physical formulae, and the code itself. A dedicated and capable novice would be fortunate to recognize 20-30%, and many months of work would be required to reach 50% recognition for codes this size. However, most code users do not require a detailed understanding of a code.

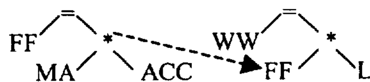
Code	Blind Test Case	Lines (k loc)	Operators (k Ops)	Semantic Declarations	Wall Time/ k Lines (s)	Quantity Recognition Rate (%)	Unit Recognition Rate (%)
flow_inlet()[9]	N	0.2	0.15	20	19.2	77.6	84.4
ALLSPD[10]	N	5.4	3.9	97	7.7	43.7	49.2
COMDES[11]	N	0.4	0.67	36	10.3	53.0	78.4
ADPAC[12]	Y	86.1	64.7	20	12.1	21.0	25.3
ENG10[13]	Y	19.7	13.3	13	7.1	16.9	13.2
PUMPA[11]	Y	1.5	2.1	31	13.2	24.0	41.1
RVCQ3D[14]	Y	3.3	3.5	18	13.5	19.6	26.4
SWIFT[15]	Y	6.6	6.6	26	29.0	21.8	30.8
STAGE2[16]	Y	4.8	6.3	32	15.6	11.8	21.3
STAGE3[17]	Y	9.8	16.4	37	24.2	20.3	33.5

Table 3: Baseline performance results for the semantic analysis program's analysis of various studied and blind test cases. Timing and recognition rate depend on semantic declarations--mainly declared array structure; the simplest semantic declarations have been used here. Calculations performed on an undedicated SGI R10000 195 MHz processor with 32 kByte data cache. The analysis results reflect the semantic analysis code's quality and not the quality or ability of the tested codes.

The Inference Tree

Analyzing the progress of expression recognition shows that recognition rates are very sensitive to inference failures and rule omissions. The inference tree represents the organization of inferences (rule recognitions) necessary to deduce a physical/mathematical expression's meaning. The inference tree is similar to the parse tree (20) of an expression except that it combines parse tree branches so that all inferences from previous assignments are present in a single tree (21) rooted at the expression's meaning.

```
C?  MA == mass, L == length
C?  ACC == acceleration
    FF = MA*ACC
    WW = FF * L
```
(20)



(21)

The inference tree (21) for variable WW contains 2 inferences and has maximum depth 2. Table 4 shows the number of inference trees versus size for each assigned variable in the 1D compressor design code COMDES.

Table 4 also shows that very large, sparse inference trees exist in scientific code. Programs typically involve many interconnected expressions so inference trees can be very large and require many successfully performed rule matches. However, since an inference (work $\Leftarrow$ force * length in (21)) is *only* possible if its predecessor branches are understood (force recognized and length defined), an error or omission will prevent subsequent inferences up to the tree root and dramatically reduce recognition. Consequently, recognition rates are very sensitive to rule errors and declaration omissions. This behavior is observed in the program examined in Table 4.

Inferences in Tree	Maximum Depth of Inference Tree (d)																										
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	>27
1	9																											
2-3		4	8																									
4-7			3	12	7																							
8-15					1	4	4	10	6	1																		
16-31							1	2	5	4	4	7	1															
32-63									1			4	3	6	6	2	2											
64-127														2	3	6	6	8	9	4	4	3	2	1				
128-255																				2	2		1	2		2	1	
256-511																							1	1	2			2

Table 4: Number of inference trees versus size ranges and tree depths. The inference trees are for each assignment in a 1D-compressor analysis code. A binary inference tree of maximum depth d can have at most $2^d - 1$ inferences; the gray diagonal represents this bound, the black region exceeds this bound, and the white region represents sparsity (not filled). These results demonstrate that inference trees can involve a large number of inferences, and be sparse.

Discussion

This automated semantic analysis procedure has properties advantageous for analysis of scientific programming. First, mathematical notation is an expressive knowledge representation. Second, parsers can use this notation and recognize mathematical, physical, and geometrical knowledge in code. Third, parsers can encapsulate formulae into independent modules. Fourth, these rules are largely fundamental, which increases generality, and they are largely aspect-independent, which reduces complexity. Last, the economics of this procedure appear to be favorable. In particular, the execution time on a modern workstation is modest. Further, the user effort to prepare semantic declarations is reasonable compared to the effort of syntactically declaring all variables as required by modern programming practice and some programming languages.

Potential Limitations

Despite these advantages, several potential limitations have been identified and must be monitored as the procedure develops. First, the rate of improvement for blind test case recognition is important and currently unknown. This rate projects the time and resources necessary to develop a practical tool; consequently, this rate determines the feasibility of this procedure. Second, as demonstrated by the inference tree analysis, the quantity recognition rate is sensitive to errors and omissions in the expert parser rules. Several other issues are of secondary concern. They include high memory requirements and undeveloped analysis capabilities (for logic, subroutine calls, the call tree, and other programming languages).

Conclusions

Scientists spend too much time slaving over their codes, analyzing details; and this experiment strives to automate these menial chores. Further, its use of fundamental representation and expert parsers provides an example for automating other scientific and engineering tasks. As detailed in the discussion section, if this procedure is to become a practical tool, knowledge rules and infrastructure must be added so that blind test case recognition rates are higher.

Acknowledgments

The lexical analysis routines and FORTRAN77 grammar are from ftnchek[18]. The GUI routines use Tcl/Tk[19]. This work was supported by the NASA High Performance Computing and Communications program through the Computing and Interdisciplinary Systems Office (contract NAS3-98008) at NASA Glenn Research Center. Greg Follen, Joe Veres, and Karl Owen were the monitors. Paul Giel, Ed Hall, Joe Veres, Kuo-Huey Chen, Rod Chima, and Karen Gundy-Burlet have provided test cases. The author thanks Andrew Appel for a great course in compilers, and Ambady Suresh and Scott Townsend for helpful discussions about this work.

Bibliography

[1]E. Kant, "Synthesis of Mathematical Modeling Software," IEEE Software, May 1993.

[2]J. Woodcock and M. Loomes, *Software Engineering Mathematics* (London: Pitman, 1988).

[3]B. D. Martino, C. W. Keßler, "Two Program Comprehension Tools for Automatic Parallelization," IEEE Concurrency, Jan-March 2000.

[4]L. M. Wills, "Automated Program Recognition: A Feasibility Demonstration," Artificial Intelligence 45 (1-2): 113-172 (1990).

[5]J. Allen, *Natural Language Understanding* (Menlo Park: Benjamin/Cummings, 1987).

[6]A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools* (Reading: Addison-Wesley, 1986).

[7]S. C. Johnson, "Yacc--Yet Another Compiler-Compiler," Comp. Sci. Tech. Rep. No. 32. (AT&T Bell Laboratories, Murray Hill, 1977).

[8]J. R. Levin, T. Mason, and D. Brown, *Lex and Yacc* (Sebastopol: O'Reilly, 1992).

[9]P. Giel, Private Communication

[10]J.-S. Shuen, K.-H. Chen, and Y. Choi, "A Coupled Implicit Method for Chemical Non-equilibrium Flows at All Speeds," J. of Comp. Phys., 106(2):306 (1993).

[11]J. Veres, Private Communication

[12]E. Hall, N. J. Heidegger, and R. A. Delaney, "ADPAC v 1.0 – User's Manual," NASA CR 1999-206600, Feb. 1999.

[13]M. E. M. Stewart, "Axisymmetric Aerodynamic Numerical Analysis of a Turbofan Engine," ASME Paper 95-GT-338, 1995.

[14]R. V. Chima, "Development of an Explicit Multigrid Algorithm for Quasi-Three-Dimensional Viscous Flows in Turbomachinery," NASA TM 87128.

[15]R. V. Chima, J. W. Yokota, "Numerical Analysis of Three-Dimensional Viscous Internal Flows," NASA TM 100878.

[16]K. L. Gundy-Burlet, M. M. Rai, R. P. Dring, "Two-dimensional computations of multistage compressors using a zonal approach," AIAA-89-2453, 1989.

[17]M. M. Rai, "Three-Dimensional Navier-Stokes Simulations of Turbine Rotor-Stator Interaction; Part 1-Methodology," *J. Propulsion and Power*, 5(3):387-396 (1987).

[18]R. K. Moniot, "ftnchek" http://www.dsm.fordham.edu/~ftnchek (Fordham University, New York, 1989).

[19]J. K. Ousterhout, *Tcl and the Tk Toolkit* (Reading: Addison-Wesley, 1994).